# Michael Chillemi

# 08/14/2023

# CS 590 - Algorithms

# M11.B4: Module 11 Graphs and Traversals Application Programming Assignment

Compared to the prior assignment, this problem didn't require as many custom classes or methods to solve. This algorithms time complexity comes out to the desired time . NetworkStations is the only class I have for this project. I have two methods in that class, printStations and reachableStations. The printStations method's objective is to give the user a clear display of all the stations as well as any additional stations they can access. In order to develop my solution, all the processing is now done in the reachableStations. A graph of all the network links and an int denoting the number of stations are inputted using this method. A vector containing a vector of integers is the result of this procedure. The number of the station will serve as the list's index, and the network stations that can connect to the current station are listed at each element of the list.

Now for the solution, I use a BFS methodology to identify the set of stations that can be reached with a maximum of four links. This technique starts with the creation of a station list, which is the list that will be output along with my solution. Next, a list of stations that can be reached, which was formerly my list and comprises connections that may be reached from each station, and an instance of a queue data structure. The method begins in a for loop and continues until the desired number of stations is reached. Every time the loop iterates, it first builds a visited list, which is a list of booleans indicating whether we've already been to that station. After all values are originally set to false, the current station on the visited list is changed to true, added to the list of values that may be reached, and then it is pushed forward in the queue. To keep track of how many links I have discovered for the current vertex, I then build a link counter. Right now, we start a while loop that will run indefinitely until the queue is empty. I then utilize the queue's pop operation after saving the current vertex as an int. After that, I perform an if statement to see if the current vertex in the visited list has been viewed and if the count is larger than 4. If both of those conditions are satisfied, we will use the break operation to exit the while loop and indicate that the current vertex has received all of the links allowed for that station.

However, in the alternative case, we first make a Boolean added\_to\_queue to show when new links for the current vertex are available. After that, we perform an if statement within a for loop that iterates across the stations. It determines whether the current iteration of the visited list hasn't been visited yet and whether the value at the graph vertex and current iteration is set equal to one. The current iteration of the visited list is also true if this is true, and the added\_to\_queue Boolean is set to true. Following that, the queue and list of stations that are reachable are updated to include the vertex. A if check on the added\_to\_queue Boolean will be performed when you exit the for loop. If it turns out to be true, the link counter will rise by one. We will add the reachable station list to the output station list once we have exited the while loop. Then, in order to have an empty queue when we return to the top of the for loop, we remove the list of stations that are accessible. The station list and the whole solution will be outputted after the main for loop is finished.

The pseudo code for reachableStations will be provided below.

Algorithm reachableStations(vector<vector<int>> graph , int n):

Input: graph, n

Output: station\_list

vector<vector<int>> station\_list

vector<int> reachable\_list

queue

for i less than n:

visted\_list[n]

for i less than n:

visted\_list[i] = false

visted\_list[i] = true

reachable\_list.push\_back(i)

queue.push\_back(i)

link\_count = 0

while !queue.isEmpty()

vertex = queue.front()

queue.pop()

if visited[vertex] == true && link\_counter > 4

break

else

added\_to\_queue = false

for v < n:

if graph[vertex][v] == 1 && !visited\_list[v]

added\_to\_queue = true

visited\_list[v] = true

queue.push(v)

reachable\_stations.push\_back(v)

if added\_to\_queue = true

link\_count++

station\_list.emplace\_back(reachable\_stations)

reachable\_stations.clear()

while !queue.isEmpty() queue.pop()

return station\_list

Regarding my test cases, which are all contained in the main.cpp file, I designed the program to be executed in one of two ways: if the user inputs 1, you will start the program in user input mode, requiring the user to manually enter all of the numbers. The application will execute hard-coded default tests if the user enters 2. A message advising the user to input one or two and restart the software will be displayed if the user enters in any other value. The test cases that Venkata gave in his most recent email are duplicated in the default test cases. I also wanted to make it clear that my work will not exactly resemble Venkata's output. However, they really are the same. Personally, I thought that having the current station as the first item in each list made it simpler to understand. This demonstrates that those test cases and my algorithm are compatible. The anticipated result will be listed below.

\*\*(Run the NetworkStations.exe in the command terminal to view the results)\*\*

$ ./NetworkStations.exe

Enter 1 for user input mode or Enter 2 for default test cases: 2

\*\* TEST CASE 1 \*\*

Reachable stations from Station 0 = {0, 1, 2, 3, 4, 5, 6, 7}

Reachable stations from Station 1 = {1, 0, 2, 3, 4, 5, 6, 7}

Reachable stations from Station 2 = {2, 1, 4, 5, 0, 3, 6, 7}

Reachable stations from Station 3 = {3, 1, 6, 7, 0, 2, 5, 4}

Reachable stations from Station 4 = {4, 2, 5, 1, 6, 0, 3, 7}

Reachable stations from Station 5 = {5, 2, 4, 6, 1, 3, 7, 0}

Reachable stations from Station 6 = {6, 3, 5, 7, 1, 2, 4, 0}

Reachable stations from Station 7 = {7, 3, 6, 1, 5, 0, 2, 4}

\*\* TEST CASE 2 \*\*

Reachable stations from Station 0 = {0, 1, 2, 3, 4}

Reachable stations from Station 1 = {1, 0, 2, 3, 4}

Reachable stations from Station 2 = {2, 1, 3, 0, 4}

Reachable stations from Station 3 = {3, 2, 4, 1, 0}

Reachable stations from Station 4 = {4, 3, 2, 1, 0}